Predavanje 3

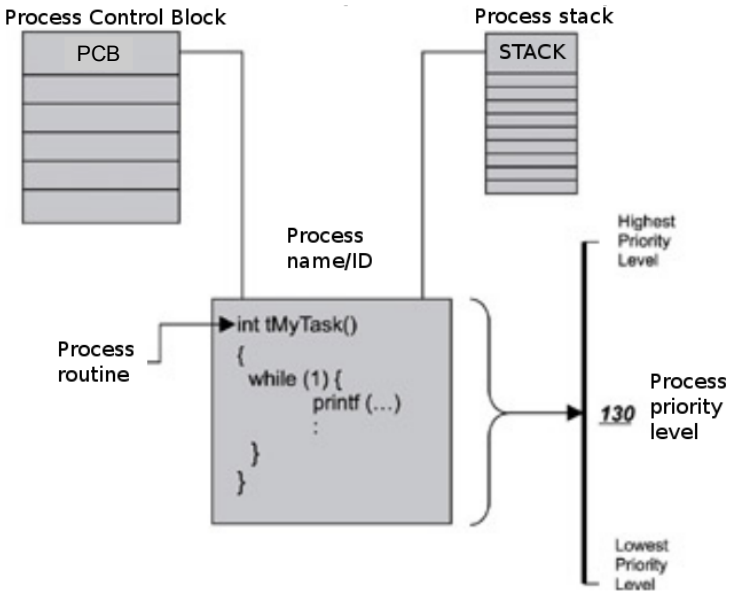
Procesi

Jednostavne softverske aplikacije se tipično dizajniraju tako da se izvršavaju sekvencijalno, jedna po jedna instrukcija, u skladu sa predefinisanim rasporedom instrukcija. Ipak, ovakva tipična šema nije prikladna za većinu real-time embedded aplikacija, koje uglavnom manipulišu sa brojnim ulazno-izlaznim signalima, često sa vremenski-ograničenim odzivom. Osim toga, real-time embedded aplikacije moraju biti dizajnirane tako da podržavaju konkurentno izvršavanje. Ovakav pristup zahteva od programera da svoje aplikacije podele u male, rasporedive, sekvencijalne blokove koda. Ukoliko se ovo korektno izvede, konkurentni dizajn omogućava multitasking izvršavanje u skladu sa očekivanim performansama i vremenskim ograničenjima real-time sistema. Većina RTOS kernela omogućava korišćenje proces objekata i upravljanje procesima u cilju implementacije konkurentnog razvoja real-time aplikacija.
Ovde ćemo poseban akcenat staviti na:

- definisanje procesa
- stanja procesa
- raspoređivanje (scheduling) procesa
- tipičnu strukturu procesa
- koordinaciju procesa
- i konkurentnost procesa.

3.1 Definisanje procesa

Proces je nezavisna nit izvršavanja koja se bori za procesorsko vreme sa konkurentnim procesima operativnog sistema. Proces kao takav mora da bude podložan raspoređivanju od strane planera. Proces je, sa druge strane, definisan svojim jedinstvenim skupom parametara i struktura podataka.
Nakon što je kreiran, svaki proces dobija jedinstven ID, dodeljuje mu se prioritet (ukoliko je on deo prioritetnog plana sa prekidanjem – *preemptive scheduling with priority*), dodeljeni PCB (process control block), stek, kao i odgovarajuće proces rutine, kao što je prikazano na slici 1. Sve gore navedene komponente čine ono što se uobičajeno naziva proces objekat u terminima operativnih sistema.



Slika 1: proces u operativnom sistemu

Kada se kernel pokreće, on kreira svoj skup sistemskih procesa i alocira odgovarajuće prioritete za svaki od njih iz skupa nivoa prioriteta. Rezervisani nivoi prioriteta se odnose na prioritete korišćene od strane RTOS-a za njegove sistemske procese. Aplikacija ne bi trebala nikako da koristi te privilegovane nivoe prioriteta koji se koriste od strane kernela, jer izvršavanje aplikacije koja koristi privilegovane nivoe prioriteta može da dovede do nestabilnog ponašanja sistema u globalu, ili čak do pada sistema. Primeri sistemskih procesa su:

- inicijalizacioni (*startup*) proces koji inicijalizuje sistem i kreira i pokreće sistemske procese
- *idle* proces koji zauzima procesorsko vreme kada nijedan drugi proces nije spreman za izvršavanje
- proces za logovanje (*logging process*) koji je zadužen za logovanje sistemskih poruka
- proces koji upravlja prekidima i izuzecima (*exceptions*)
- debug agent koji omogućava debagovanje sistema korišćenjem host debuger-a.

Naravno, dodatni sistemski procesi mogu biti kreirani naknadno tokom inicijalizacije, u zavisnosti od toga koji moduli i komponente se uključuju u kernel. Idle proces, koji se kreira prilikom podizanja kernela je sistemski proces koji svakako zahteva posebnu pažnju iako deluje da je najnepotrebniji. On je uvek postavljen kao proces sa najniži prioritetom, tipično se izvršava u beskonačnoj petlji i aktivan je uvek kada nijedan drugi proces nije aktivan, ili nijedan drugi proces nije uopšte kreiran, a sve u cilju da koristi procesorsko vreme i drži procesor “zauzetim”. Ovaj proces je neophodan jer procesor u svakom trenutku izvršava instrukciju na koju pokazuje programski brojač. Osim u slučaju kada procesor može biti u suspendovanom stanju, programski brojač u svakom trenutku mora da pokazuje na validnu instrukciju koja će se izvršavati. U skladu sa tim, idle proces obezbeđuje da je sadržaj programskog brojača uvek validan čak i kada ne postoji nijedan proces koji se izvršava.

Postoje, ipak, slučajevi kada kernel omogućava korisničkim rutinama da budu pokrenute umesto idle procesa u cilju implementiranja specijalnih zahteva za specifične aplikacije. Jedan primer bi bio mod u kome je cilj smanjenje potrošnje sistema. Tada, ukoliko nijedan drugi proces nije aktivan i ne može da se izvršava, kernel prepušta kontrolu korisničkoj rutini umesto idle procesu. U ovom scenariju, korisnička rutina tipično umesto beskonačne petlje inicira odlazak u režim izuzetno niske potrošnje (suspendovani sistem ili slično) nakon kraćeg “idle” perioda.

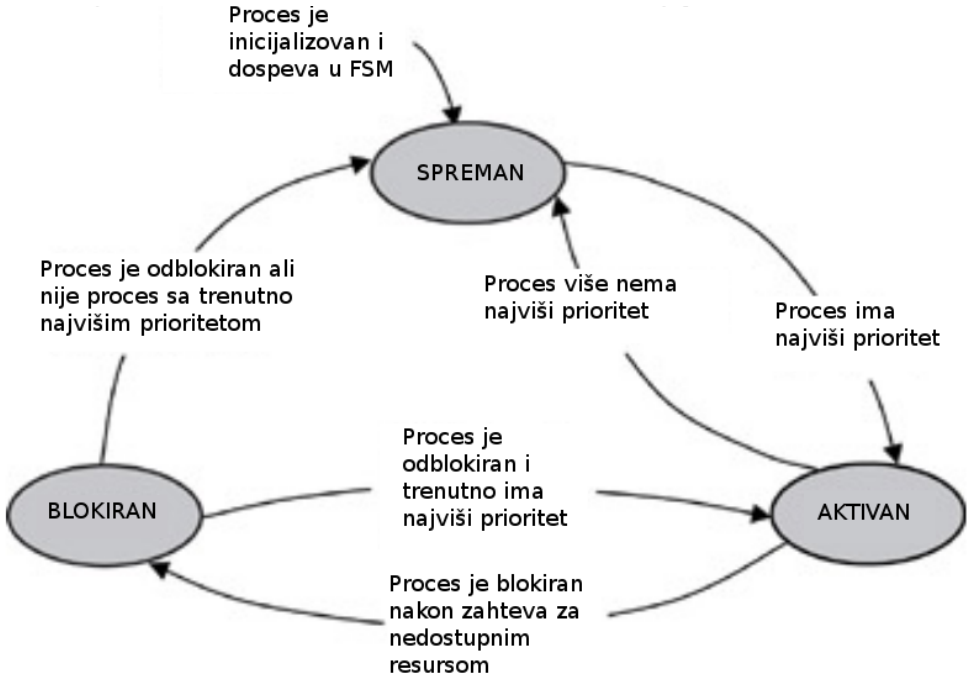
Nakon što se kernel inicijalizuje i kreira sve potrebne i neophodne procese, kernel skače na predefinisanu ulaznu tačku (obično predefinisana funkcija) koja služi kao početak aplikacije. Počevši od te ulazne tačke, korisnik inicijalizuje i kreira ostale aplikacione procese, kao i ostale kernel objekte, eventualno zahtevane od strane aplikacije.

Nakon kreiranja svakog pojedinačnog procesa, korisnik mora dodeliti svakom procesu ime, prioritet, veličinu steka, kao i samu rutinu koja treba da se izvršava u okviru procesa. Ostatak posla se obavlja od stane kernela, pri čemu kernel dodeljuje svakom procesu jedinstven ID i kreira odgovarajući PCB (*Process Control Block*) za svaki od procesa, kao i zahtevani nezavisni stek prostor za svaki proces.

3.2 Stanja procesa i veza sa planerom

Bez obzira da li se radi o sistemskom ili aplikacionom procesu, u datom vremenskom trenutku proces se nalazi u jednom od predefinisanih stanja: spreman (*ready*), aktivan (*running*) ili blokiran (*blocked*).

Tokom rada real-time operativnog sistema, svaki proces prelazi iz jednog stanja u drugo u skladu sa logikom jednostavnog konačnog automata (FSM - Finite State Machine). Slika 2 ilustruje tipičnu FSM za stanje prilikom izvršavanja procesa, sa kratkim opisima prelaza iz stanja u stanje.



Slika 2: Stanja procesa

Iako kernel može da definiše drugačija stanja i podele stanja, generalno, tri osnovna stanja se koriste u slučaju većine tipičnih kernela sa prioritetnim planiranjem i prekidanjem:

- *stanje SPREMAN*

proces se nalazi u ovom stanju kada je spreman za izvršavanje ali nije aktivan jer je proces višeg prioriteta trenutno onaj koji se izvršava.

- *stanje BLOKIRAN*

proces se nalazi u blokiranom stanju kada zahteva sistemski resurs koji mu u tom trenutku nije na raspolaganju, ili proces jednostavno čeka na neki događaj (*event*) da bi nastavio sa svojim radom, ili je jednostavno rešio da sačeka određeni period vremena pre nego što nastavi sa radom (npr. mora da sačeka na rezultat A/D konverzije pre nego što krene da obrađuje ulazni signal).

- *stanje AKTIVAN*

proces se nalazi u ovom stanju ukoliko ima najviši prioritet i trenutno je spreman za izvršavanje.

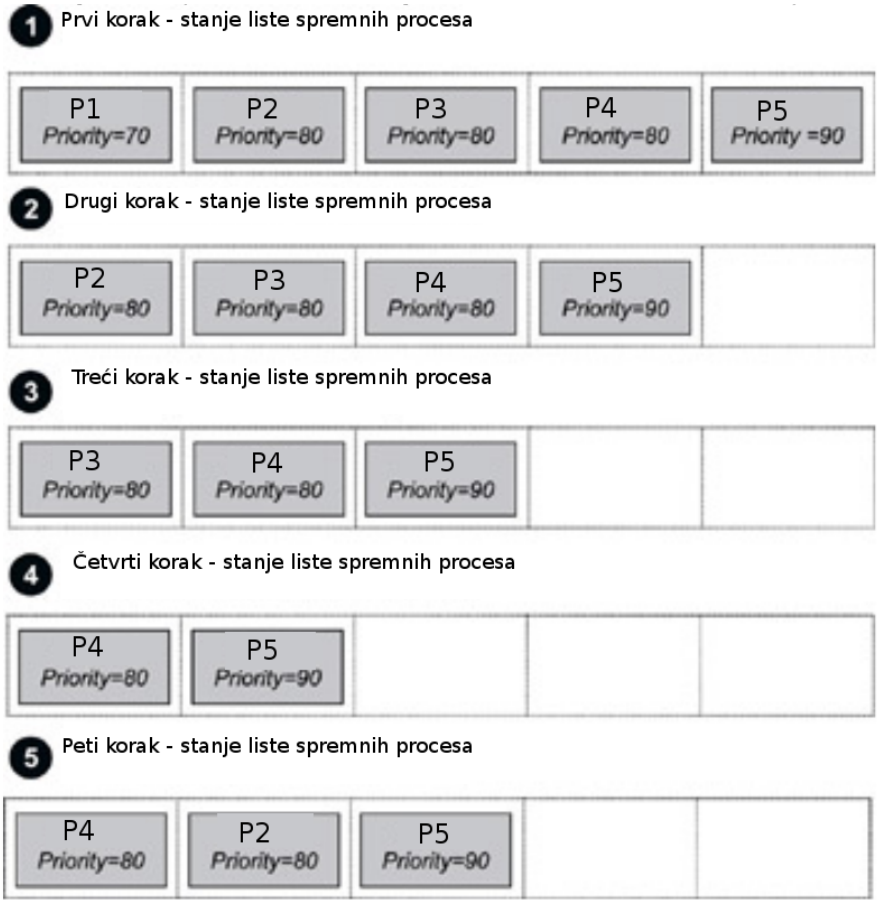
Treba napomenuti da neki, pre svega komercijalni RTOS-i implementiraju i dodatna stanja u kojima se može naći proces: suspendovano stanje, odloženo stanje itd. U ovakvim slučajevima ova dodatna stanja su uglavnom samo funkcionalna podstanja nekog od tri osnovna stanja (npr. blokirano stanje može da nastane kada proces čeka neki resurs koji mu trenutno nije dostupan što odgovara takozvanom *pending* stanju, dok be *delayed* stanje odgovaralo situaciji kada proces čeka da istekne određeni period vremena kako bi nastavio sa radom). Suspendovano stanje se uglavnom koristi za debagovanje.

Ipak, nezavisno od toga kako kernel implementira konačni automat stanja u kojima se nalazi, kernel mora da kontroliše i održava stanja svih procesa koji postoje u sistemu. Nakon svakog sistemskog poziva od strane procesa koji se trenutno izvršava, kernel, odnosno planer najpre odlučuje koji procesi treba da promene stanje i onda izvršava te promene.

Treba obratiti pažnju da postoje slučajevi kada kernel menja stanja određenih procesa, ali to ne rezultuje zamenom konteksta pošto se stanje procesa sa najvišim prioritetom nije promenilo. U svim ostalim slučajevima, pak, promena stanja nekog od procesa imaće za rezultat zamenu konteksta pošto je prethodni proces sa najvišim prioritetom ili prešao u blokirano stanje ili jednostavno više nije proces sa najvišim prioritetom među procesima koji su u tom trenutku spremni (nalaze se u stanju SPREMAN). Kada se to desi, prethodno aktivni proces se stavlja u *blokirano* stanje ili postaje *spreman*, a novi proces sa najvišim prioritetom postaje *aktivan*. U nastavku će biti objašnjena stanja procesa detaljnije. Treba skrenuti pažnju, ipak, da se opis koji sledi odnosi na sistem sa jednim procesorskim jezgrom, na kome se izvršava kernel sa prioritetnim algoritmom planiranja sa prekidima.

3.2.1 Stanje SPREMAN

Kada se proces kreira i inicijalizuje tako da je spreman za izvršavanje, kernel ga postavlja u stanje *spreman*. U ovom stanju proces se aktivno bori sa ostalim procesima za procesorsko vreme. Kao što je prikazano na slici 2, proces iz ovog stanja ne može biti direktno premešten u *blokirano* stanje. Proces mora prethodno proći kroz *aktivno* stanje u kojem će se izvršavati. Dok je u *aktivnom* stanju, proces može izvršiti poziv blokirajuće funkcije, tj. funkcije koja ne može da se u potpunosti izvrši u datom trenutku, i kao rezultat toga, proces će se premestiti na listu blokiranih procesa.
Dakle, proces koji je u stanju *spreman*, može samo preći u stanje *aktivan*, u skladu sa konačnim automatom predstavljenim na slici 2. Obzirom na činjenicu da mnogo procesa u datom trenutku može da se nalazi u stanju *spreman*, kernelov planer koristi prioritete kako bi odlučio koji će proces sledeći biti *aktivan*. Za kernel koji dozvoljava samo jedan proces za svaki nivo prioriteta koji podržava, ovaj posao je prilično jednoznačan i nedvosmislen: proces sa najvišim prioritetom u datom trenutku dobija procesorsko vreme. U ovakvoj implementaciji, kernel limitira broj procesa na broj dozvoljenih nivoa prioriteta. Ipak, većina kernela omogućava više od jednog procesa po nivou prioriteta, pri tome dozvoljavajući i veći broj procesa u sistemu uopšte. U ovakvom scenariju, algoritam planiranja je komplikovaniji i kompleksniji i uključuje održavanje *liste spremnih procesa* - liste procesa koji su u stanju *spreman*. Neki kerneli čak implementiraju posebnu listu spremnih procesa za svaki nivo prioriteta, dok ostali implementiraju kombinovanu listu. Slika 3 ilustruje scenario od 5 koraka, u kome je prikazano kako planer potencijalno koristi listu spremnih procesa kako bi premeštao procese iz stanja *spreman* u stanje *aktivan*. Ovaj primer podrazumeva da se koristi procesor sa jednim jezgrom (*single-core*) i prioritetni algoritam za planiranje sa prekidanjem u kojem je 255 najniži a 0 najviši nivo prioriteta. U cilju pojednostavljenja, u ovom primeru nisu prikazani sistemski procesi (kao npr *idle* proces).



Slika 3: Održavanje liste spremnih procesa od strane kernela

U gore prikazanom primeru, procesi 1, 2, 3, 4 i 5 su spremni za izvršavanje, i kernel ih postavlja u red čekanja u skladu sa njihovim prioritetima. Proces 1 ima najviši prioritet (70), procesi 2,3 i 4 imaju sledeći najviši prioritet (80), dok je proces 5 sa najnižim prioritetom (90). Sledeći koraci opisuju kako kernel koristi listu procesa da bi prebacivao procese u i iz stanja spreman.

1. Procesi 1, 2, 3, 4, i 5 su spremni za izvršavanje i čekaju u listi čekanja

2. Pošto proces 1 ima najviši prioritet, on je prvi proces koji se uzima iz liste i on će biti prvi proces koji se premešta u aktivno stanje. Ukoliko se u datom trenutku ne izvršava nijedan proces višeg prioriteta, kernel uklanja proces 1 iz liste i premešta ga u aktivno stanje (proces se izvršava).

3. Tokom izvršavanja, proces 1 pravi blokirajući poziv, tj. poziva blokirajuću funkciju. Kao rezultat, kernel premešta proces 1 u blokirajuće stanje, uzima proces 2 koji je prvi sledeći u redu čekanja po nivou prioriteta i postavlja proces 2 u aktivno stanje.

4. Nakon toga, proces 2 poziva blokirajuću funkciju. Kernel premešta proces u blokirajuće stanje, uzima proces 3 koji je sledeći po redu prioriteta i premešta njega u stanje u kome će se izvršavati.

5. Dok se proces 3 izvršava, oslobađa resurse koji su bili zauzeti i zbog kojih je proces 2 otišao u blokirajuće stanje. Kao rezultat, kernel vraća proces 2 u listru spremnih

procesa i dodaje ga na kraj reda čekanja procesa sa nivoom prioriteta 80. Proces 3 nastavlja da se izvršava.

Iako nije prikazano ovde, ukolio bi proces 1 postao odblokiran u nastavku, tokom izvršavanja procesa 3, kernel bi premestio proces 1 u stanje *aktivan*, obzirom na činjenicu da je njegov prioritet viši od procesa koji se trenutno izvršava (proces 3). Kao u slučaju procesa 2 od ranije, proces 3 u ovom momentu bi bio premešten u listu spremnih procesa odmah iza procesa 2, koji ima isti prioritet kao i on i ispred procesa 5 sa nižim prioritetom.

3.2.2 Aktivno stanje procesa

Kod sistema sa jednim procesorom, samo jedan proces može da se izvršava u datom trenutku. Kada se proces prebaci u aktivno stanje, kao što smo već pominjali ranije, procesor učitava vrednosti sačuvanih registara tog procesa iz njegovog konteksta. Nakon toga, procesor može da nastavi da izvršava instrukcije definisane procesom i upravlja njegovim stekom. Kao što je prethodno spomenuto, kada se proces premešta u stanje *spreman*, to znači da je izvršavanje procesa prekinuto od strane procesa sa višim prioritetom.
Za razliku od procesa u stanju spreman, *aktivan* proces može da se premesti u blokirajuće stanje ako se desi bilo koji od sledećih uslova:
- zahteva se resurs koji je trenutno nedostupan
- vrši se zahtev za čekanjem na određeni događaj koji treba da se desi
- vrši se zahtev za privremenim odlaganjem izvršavanja procesa

U svakom od ovih slučajeva, proces se premešta iz *aktivnog* stanja u *blokirajuće* stanje.

3.2.3 Blokirajuće stanje procesa

Mogućnost postavljanja procesa u blokirajuće stanje je izuzetno značajna za samu funkcionalnost real-time sistema jer, u nedostatku istog, procesi sa nižim prioritetom se nikada ne bi izborili za procesorsko vreme, i samim tim, nikada se ne bi izvršavali. Ukoliko procesi sa višim prioritetom nisu dizajnirani da koriste blokirajuće stanje, oni mogu da zauzmu CPU vreme što će dovesti do problema izgladnjivanja (*starvation problem*) opisanog ranije. Proces može da pređe u blokirajuće stanje samo nakon blokirajućeg poziva, zahtevajući postojanje blokirajućeg uslova da bude ispunjen (ovaj uslov bi se verovatno trebao zvati od-blokirajući uslov, ali termin blokirajući se odomaćio kod programera i redovno se koristi).
Primeri blokirajućih uslova bi bili:

- semafor na koji čeka blokirani proces je oslobođen
- poruka na koju čeka proces je pristigla u red poruka, ili je jednostavno istekao time-out konfigurisan za proces prilikom čekanja te poruke

Kada se proces odblokira, kernel ga premešta u listu spremnih procesa, ukoliko nije proces sa najvišim prioritetom u datom trenutku. Ukoliko odblokirani proces ima najviši prioritet, on se automatski premešta u aktivno stanje (bez prolaska kroz listu spremnih procesa) i njegov kontekst zamenjuje kontekst procesa koji se u tom trenutku izvršava. Proces koji je prekinut u takvom scenariju se premešta u listu spremnih procesa na odgovarajuću poziciju u skladu sa njegovim prioritetom.

3.3 Osnovne operacije na procesima

Osim samog korišćenja procesa, kernel obezbeđuje i specijalne mehanizme upravljanja i kontrole procesa. One uključuju akcije koje kernel obavlja “iza scene” kako bi odradio specifične operacije na procesima: npr. kreiranje procesa, održavanje PCB bloka procesa ili steka procesa. Kernel, osim ovoga, obezbeđuje API kako bi korisnici takođe mogli da manipulišu procesima. Neke od najtipičninih operacija koje programeri mogu da izvršavaju na proces objektima sa strane aplikacije uključuju:

- kreiranje i brisanje procesa
- kontrola planiranja procesorskog vremena i stanja u kome se proces nalazi
- dobijanje informacija o procesima.

Za svaki projekat koji se radi, i svaki kernel koji se koristi, programer mora biti upoznat sa ovim elementarnim operacijama za manipulisanje procesima. Svaka od njih će u nastavku biti detaljnije objašnjena.

3.3.1 Kreiranje i brisanje procesa

Ovo su svakako najosnovnije operacije koje programer mora da nauči pre nego što uopšte počne sa programiranjem u datom RTOS-u.

Operacija	*Opis*
Create	Kreira proces u okviru kernela
Delete	Briše proces iz konteksta kernela

Tabela 1: Operacije za kreiranje i brisanje procesa

Programeri obično kreiraju proces korišćenjem jedne ili dve operacije. Neki kerneli omogućavaju kroz svoj API funkciju za kreiranje, i funkciju za pokretanje procesa. U ovakvom scenariju, proces nakon što se kreira se stavlja u suspendovano stanje, a kasnije se premešta u stanje *spreman* kada bude startovan (spreman za izvršavanje). Kreiranje procesa na ovaj način može biti korisno ukoliko se koristi u svrhu debagovanja ili kada je potrebno obaviti neke specijalne inicijalizacione operacije između trenutka kreiranja procesa i trenutka kada će on biti pokrenut. Ipak, u većini slučajeva, dovoljno je samo kreirati i pokrenuti proces jednom API funkcijom. Suspendovano stanje je slično blokirajućem stanju u smislu da proces koji je suspendovan nije aktivan, ali svakako nije ni spreman za izvršavanje (nije u stanju *spreman*). Međutim, proces neće ući u suspendovano stanje, niti izaći iz njega, ukoliko se dese događaji koji neki proces mogu postaviti u blokirajuće stanje, ili ga premestiti iz blokirajućeg u neko drugo stanje. Prava svrha i način korišćenja suspendovanog stanja, ukoliko ono uopšte postoji, varira od RTOS-a do RTOS-a.
Pokretanje procesa ne znači da će on biti aktivan momentalno, to samo znači da će proces biti smešten u odgovarajući red čekanja (listu spremnih procesa). Ipak, većina kernela omogućava konfigurabilne ulazne tačke (eng. *hooks*), koje zapravo predstavljaju mehanizam izvršavanja korisničkih funkcija, u kritičnim momentima dešavanja specifičnih događaja unutar kernel-a. Ove ulazne tačke omogućavaju programeru da registruje fukciju u okviru kernela prosleđujući pokazivač na funkciju API metodi definisanoj od stane kernel-a. Kernel će pokrenuti izvršavanje te funkcije

kada se desi događaj za koji je data funkcija registrovana. Ovi događaji su najčešće:

- trenutak kada je proces kreiran
- trenutak kada je proces premešten u suspendovano stanje, blokirajuće stanje ili kada se iz bilo kog razloga desila zamena konteksta
- kada je proces obrisan

Ove ulazne tačke su veoma korisne kada se, recimo, prati status nekog od procesa kako on prolazi kroz stanja, ili se recimo vrši kontrola stanja procesa u trenutku zamene konteksta, ili prilikom brisanja procesa.
Programer svakako treba da bude veoma oprezan prilikom brisanja procesa u nekom embedded sistemu. Većina kernela dozvoljava brisanje nekog od procesa iz nekog drugog procesa. Prilikom procesa brisanja, kernel zapravo terminira proces i oslobađa memoriju koju je taj proces-koristio (PCB i stek). Međutim, kada se proces izvršava, oni najčešće zauzmu veću količinu memorije ili drugih kernel-objekata. Ukoliko se proces obriše nepažljivo, može doći do narušavanja celog sistema. Na primer, pretpostavimo da proces zauzme semafor kako bi imao ekskluzivno pravo korišćenja deljene memorije ili neke strukture podataka. Dok proces izvršava operacije na toj strukturi podataka (ili jednostavno bloku memorije), proces se obriše. Ukoliko se ovo ne uradi pažljivo može doći do:

- postojanja strukture podataka koja nije “validna” nakon što se proces obriše (usled nekompletirane operacije upisa u strukturu);
- neoslobođenog semafora, koji više nije na raspolaganju ostalim procesima koji ga koriste u sistemu (neki drugi proces koji čeka na taj semafor, više nikada neće biti pozvan);
- nepristupačna struktura podataka koja više nije dostupna nekom drugom procesu usled činjenice da semafor koji kontroliše pristup nije oslobođen.

Kao rezultat, prevremeno brisanje procesa može rezultovati “*curenjem*” memorije ili resursa (*memory leak, resource leak*).
Curenje memorije nastaje dovodi do trenutka kada će kompletan sistem ostati bez memorije (pre ili kasnije), dok curenje resursa, osim same nemogućnosti korišćenja resursa, dovodi do sličnog ostatka bez memorije jer svaki od kernel-objekata (npr. semafor) zauzima određenu količinu memorije sam po sebi.
Većina kernela programerima daje na raspolaganje takozvano “zaključavanje” prilikom brisanja procesa, što zapravo predstavlja par funkcijskih poziva koji onemogućava brisanje procesa prilikom izvršavanja kritične sekcije koda. Kasnije ćemo više pažnje posvetiti ovome. Ovde je samo važno istaći da svaki proces koji se briše iz kernela, mora imati dovoljno vremena na raspolaganju kako bi očistio za sobom i oslobodio resurse i memoriju koje je koristio.

3.3.2 Kontrola planiranja procesorskog vremena i stanja u kome se proces nalazi

Od trenutka kreiranja procesa do trenutka kada je proces obrisan, proces uobičajeno prolazi kroz mnogobrojna stanja u skladu sa tokom izvršavanja aplikacije i rasporeda izvršavanja procesa od strane planera. Iako se uglavnom promena stanja vrši automatski (od strane planera), većina kernela omogućava skup API funkcija kojima programer može da utiče na tok izvršavanja i promenu stanja svojih procesa. Tabela 2 prikazuje ove funkcije.

Operacije	*Opis operacije*
Suspend	Suspendovanje procesa
Resume	Nastavak izvršavanja procesa iz suspendovanog stanja
Delay	Odlaganje izvršavanja procesa
Restart	Pokretanje procesa od početka
Get priority	Dobijanje informacije o trenutnom prioritetu procesa u okviru kernela
Set priority	Postavljanje prioriteta procesa
Preemption lock	Trenutno zabranjuje procesima višeg prioriteta prekid izvršavanja trenutno izvršavanog procesa
Preemption unlock	Dozvola procesima višeg prioriteta prekid izvršavanja trenutno izvršavanog procesa

Tabela 2: Kontrola planera i stanja procesa

Korišćenjem ovih funkcija za manipulaciju planerom, programer može da suspenduje ili nastavi izvršavanje procesa iz aplikacije i ovo se najčešće vrši prilikom debagovanja, ili na primer trenutnog suspendovanja procesa sa visokim prioritetom, kako bi se mogli izvršiti (barem delimično) procesi nižeg prioriteta.
Sa druge strane, programer možda želi da odloži izvršavanje procesa na određeno vreme i na taj način ga premesti u blokirajuće stanje određenog trajanja, u cilju čekanja na neki eksterni događaj kojem nije dodeljena prekidna rutina. Samim prekidanjem izvršavanja procesa, oslobađa se procesor, i omogućava se drugom procesu da počne sa izvršavanjem. Nakon što istekne period odlaganja, proces se vraća u listu spremnih procesa na kraj dela reda sa procesima istog prioriteta. Proces kojem je privremeno odloženo izvršavanje funkcijom *Delay*, uglavnom se probudi i proveri da li se čekani događaj desio, ili ne. Ukoliko nije, novo odlaganje izvršavanja sledi (*polling*).
Programer često ima potrebu da u određenom trenutku ponovo počne sa izvršavanjem procesa od samog početka (*Restart*), što je svakako drugačije od samog nastavka rada nakon što je proces bio suspendovan (*Resume*). Pokretanje procesa od početka znači da će proces biti tretiran kao da je upravo kreiran i stavljen na raspolaganje kernelu. Kompletan kontekst procesa koji je on imao prilikom pozivanja operacije Restart, će biti izgubljen, nasuprot operaciji Resume, u slučaju koje proces zadržava svoj kontekst i sve resurse koje je proces prethodno zauzeo/registrovao. Restart procesa je najčešće koristan prilikom debagovanja ili kada je neophodna reinicijalizacija nekog procesa nakon ozbiljne greške nastale tokom izvršavanja programa.
Proveravanja i postavljanje prioriteta procesa tokom izvršavanja aplikacije omogućava programeru da utiče na promenu rasporeda izvršavanja procesa dinamički. Ovaj mehanizam je neophodan u slučaju potreba izvršenja “zamene prioriteta” kada proces sa nižim prioritetom zauzima deljeni resurs koji se čeka od strane procesa sa višim prioritetom, čijim blokiranjem je pokrenut proces srednjeg prioriteta. Jednostavno rešenje ovakvog problema je oslobađanje zahtevanog resursa dinamičkim povećavanjem prioriteta procesa niskog prioriteta (koje je neophodno kako bi se taj proces uopšte izborio za procesorsko vreme pored procesa sa srednjim prioritetom), nakon čega sledi vraćanje prioriteta procesa na prethodnu “nisku” vrednost.
Na kraju, kernel omogućava zabranu/dozvolu prekidanja izvršavanja nekog procesa od strane procesa sa višim prioritetom. Ovo je najčešće neophodno u trenucima kada proces niskog prioriteta prolazi kroz *kritičnu sekciju koda*.

3.3.3 Dobijanje informacija o procesima

Kernel omogućava rutine kojima programeri mogu da dobijaju informacije o stanju procesa tokom izvršavanja aplikacije kao što je pokazano u tabeli 3. Ove rutine su korisne za debagovanje i monitorisanje stanja sistema.

Operacija	*Opis*
Get ID	Čitanje identifikacionog broja trenutnog procesa
Get PCB	Dobijanje PCB-a željenog procesa

Tabela 3: Metode za dobijanje informacija o procesima

Najčešće korišćeno od strane aplikacija, nakon dobijanja identifikacionog broja nekog od procesa, korišćenjem tog ID-ja dobija se PCB određenog procesa (preko njegovog ID-ja). Čitanjem PCB-a, sa druge strane, dobija se samo "trenutna" slika stanja procesa. Ukoliko proces nije u suspendovanom ili blokiranom stanju, njegov kontekst se dinamički menja, te se sukcesivnim pozivanjem metode *Get PCB* dobijaju drugačiji podaci. Ovo treba imati u vidu i oprezno koristiti metode iz Tabele 3, kako se ne bi donosile odluke bazirane na zastarelom stanju procesa koji konstantno menja svoj kontekst, tokom izvršavanja aplikacije.

3.4 Tipična struktura procesa

Prilikom pisanja koda, procesi su obično struktuirani na jedan od dva načina:

1. procesi koji se izvršavaju do završetka
2. beskonačne petlje

Obe strukture su prilično jednostavne. Procesi koji se izvršavaju do završetka su najčešće korišćeni za inicijalizaciju, obično se izvršavaju samo jednom i to prilikom uključenja napajanja i podizanja sistema. Sa druge strane, procesi realizovani kroz beskonačne petlje se obično koriste za većinu posla oko manipulisanja ulazima i izlazima, procesiranje primljenih podataka i slično.

Pseudo kodovi za tipične dve strukture su prikazani ispod:

```
RunToCompletionProcess ()
{
     Inicijalizacija aplikacije
     Kreiraj procese sa beskonačnom petljom
     Kreiraj kernel-ove objekte
     Obriši ili suspenduj ovaj proces
}
```

Inicijalizacioni proces obično ima viši prioritet u poređenju sa procesima koje on kreira, tako da se inicijalizacija ne može prekinuti u toku izvršavanja. U najjednostavnijem obliku, ostali procesi su jedna ili više rutina koje se izvršavaju u beskonačnoj petlji. Inicijalizacioni proces se obično kreira tako da se na kraju obriše ili suspenduje nakon što odradi neophodne inicijalizacione rutine. Nakon toga, novokreirani procesi mogu da se nesmetano izvršavaju.

Primer procesa koji se izvršava u beskonačnoj petlji je dat ispod. Slično kao i u prošlom primeru, i

ovi procesi mogu da sadrže deo koji se odnosi na inicijalizaciju, međutim, ovaj deo koda se izvršava samo kada se proces prvi put izvršava, nakon čega se počne izvršavati beskonačna petlja i periodično sve rutine koje su u nju uključene. Kritičan deo svakog procesa koji je implementiran kao beskonačna petlja jeste postojanje ***barem jednog blokirajućeg poziva*** u okviru tela petlje. Ovo omogućava izvršavanje procesa sa nižim prioritetom.

```
EndlessLoopProcess ()
{
     Inicijalizacioni kod
     While(1)
     {
          Telo petlje
          Poziv jedne ili više blokirajućih rutina
     }
}
```

3.4 Sinhronizacija, inter-komunikacija i konkurentnost procesa

Procesi se sinhronišu i komuniciraju između sebe korišćenjem inter-proces komunikacionih metoda (Inter Process Communication-IPC), koje su zapravo kernel - objekti. Oni omogućavaju sinhronizaciju i komunikaciju između dva ili više niti (thread) koje se izvršavaju u “paraleli”.
Primeri takvih objekata su semafori, redovi poruka, signali, cevi (pipes), itd. Svaki od ovih objekata će biti detaljnije analiziran u nastavku.
Koncept konkurentnosti i načina na koji se aplikacija segmentira u konkurentne procese će takođe biti diskutovan u nastavku.