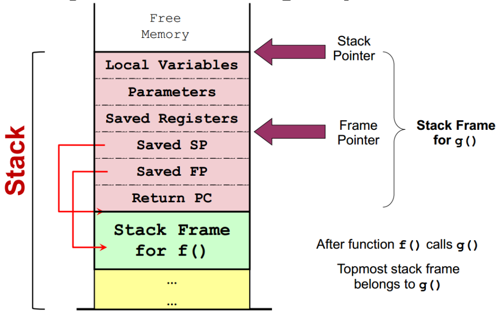
1 Introduction to OS

OS: intermediary between user and hardware
- time-sharing: illusion of concurrency
- allocate resources, control program
- Kernel; monolithic vs. microkernel has more overhead but smaller
- Hypervisors: T1 (over hardware), T2 (over host OS)

2 Process Abstraction

Process: abstraction of running program
- memory/ hardware/ OS context

Stack Memory Region
- Stack frame (sf): for function invocation
- stack grows → address decreases
- Setup/ teardown managed by OS

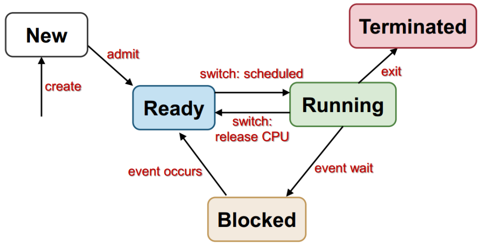


- Frame pointer: fixed location in sf
- Saved registers; GPRs may be temporarily stored in memory first (register spilling)

- On executing function call:
  - **Caller**: Pass arguments with registers and/or stack
  - **Caller**: Save Return PC on stack
  - **Transfer control from caller to callee**
  - **Callee**: Save registers used by callee. Save old FP, SP
  - **Callee**: Allocate space for local variables of callee on stack
  - **Callee**: Adjust SP to point to new stack top
- On returning from function call:
  - **Callee**: Restore saved registers, FP, SP
  - Transfer control from callee to caller using saved PC
  - **Caller**: Continues execution in caller

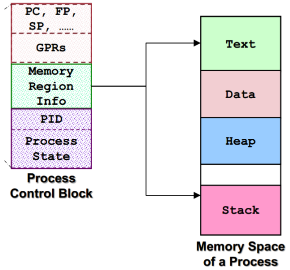
Memory Context
- Heap: dynamically allocated memory (malloc, new)

OS Context
- Process identification (PID) – reuse? Limit to maximum no.? Reserved?
- Process state



Hardware Context - GPR, PC, SP, FP

Process Table
- Contains PCBs



- Text: Instructions, Data: Global var
- Pointers are in stack, but memory region pointed to can be in heap

Interaction with OS
- System Calls: API to OS (involves change to kernel mode, via TRAP), dispatcher finds appropriate system call handler
- Function wrapper (same)/ adapter (modified)
- Exceptions (synchronous, due to program execution) and Interrupts (async, e.g. CTRL-C/ kill)

UNIX Case Study
- init: root process of process tree
- fork(): creates duplicate process
  . returns PID of child
  . can exec() another process
  . exit(), wait(&status), wait(NULL)
  . Copy on Write: only duplicate memory when changed
- Zombie processes:
  . Parent exists before child: init becomes pseudo parent
  . Child terminates before wait: child becomes zombie (PCB not cleared)

3 Process Scheduling

Concurrency/ pseudoparallelism
- Has to be fair, balance of resources
- Scheduler triggered (timer interrupts), decides if context switch is needed
- Cooperative vs. preemptive (fixed time quota, time slicing)

Batch Processing
- No user interaction, non-preemptive
- Turnaround time: finish – arrival time
- Throughput: #tasks/ unit time
- CPU utilisation: %time CPU busy

First Come First Serve
- FIFO, no starvation
- Convoy effect: long process runs first

Shortest Job First
- Minimises average waiting time
- Starvation possible
- Need to know total CPU time for a task (predicted using exponential average)

Shortest Remaining Time
- New job with shorter remaining time can preempt current running job
- Good service for short jobs, even with late arrivals

Interactive Systems
- Response time: response – request time
- Predictability (variance in response time)
- Preemptive: scheduler runs periodically)
- Interval of timer interrupt (ITI)
- Time quantum: multiple of ITI
- Context switches when time quanta over
- Remaining time quantum: next process can be scheduled

Round Robin
- FIFO with fixed time slice
- Response time guarantee: (n-1)*q
- Larger quantum: more CPU utilisation, but longer response time

Priority Scheduling
- Variation; higher priority process can optionally preempt lower priority process
- P1 = highest
- Low priority process can starve (can decrease priority after every time quantum)

Priority Inversion: P1 process depends on resource locked by P3, P2 runs instead

Multi-Level Feedback Queue (MLFQ)
- Scheduling without perfect knowledge, minimises response time for I/O bound processes and turnaround time for CPU-bound processes
- Priority reduced if job fully utilises time slice
  . Counter abuse using cumulative time
  . Periodically shift to highest priority

Lottery Scheduling
- Can change % of tickets owned
- Responsive: new process has a chance to run

4 Inter-Process Communication

Shared Memory

- P1 creates M, P2 attached M to its own memspace (synchronisation problems)
- UNIX: shmget, shmat, shmdt, shmctl

Message Passing

- Msg in kernel memory space
- Direct: explicitly name other party (must know identity)
- Indirect: Mailbox; shared among multiple processes
- Synchronous: blocking

UNIX Pipes

- Input, output, error
- Circular bounded buffer
- Data accessed in FIFO order
- Producer-Consumer relationship
- dup2 for input/ output redirection

UNIX Signals

- Async (interrupts); kill, stop, continue, errors…

5 Threads

- Lightweight; share code, data and files
- Duplicated registers and stack (SP, FP points to different location in same main stack)
- More resource sharing, responsive, scalable, less overhead, less protection
- Can execute different threads in parallel
- fork(): usually only one thread. Exit()? Exec()? Which thread handles signal?
- User thread: library, kernel unaware
    . One thread blocks → whole process blocks
- Kernel thread: can have thread-level scheduling
- POSIX: pthread (-lpthread)
    . Shared memory space
    . pthread_join to synchronise

6 Synchronisation

Race Condition

- Non-atomic instructions, outcome depends on order of execution

Critical Section (CS)

- Mutual exclusion, progress, bounded wait, independence

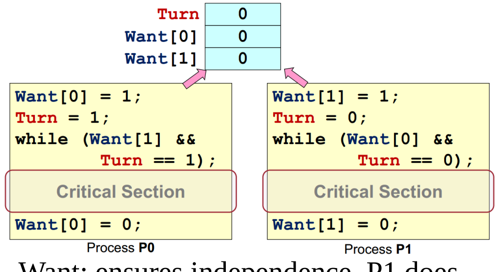
Problems

- Deadlock: all blocked
- Live Lock: no progress due to deadlock avoidance mechanism
- Starvation: some processes blocked forever

TestAndSet

- Atomic, assembly-level
- Load content at memory location into register, set memlocation = 1
- Busy waiting (processes still scheduled)
- Bounded wait: depends on scheduler

Peterson Algorithm



- Want; ensures independence, P1 does not have to wait for P0 to enter CS first
- Turn;
    . Ensures mutual exclusion: only holds one value at each time
    . Prevents deadlock: both "want" = 1
- Busy waiting
- Only synchronises 2 processes

Semaphore

- Atomic wait, signal
- Wait: s <= 0: block; s--;
- Signal: s++; wakes up one sleeping process (if any)
- *Invariant*: $S_{current} = S_{initial}$ + #signal(s) - #wait(s), wait() must be completed
- Binary semaphore: mutex
- #in CS = #wait(s) - #signal(s) <= 1
- No deadlock ($S_{current}$ and N_{CS} cannot both be 0)
    . Unless two semaphores blocked
- No starvation, assuming fair scheduling

Producer-Consumer (e.g pipe)

```
while (TRUE) {
        Produce Item;

        wait( notFull );
        wait( mutex );
        buffer[in] = item;
        in = (in+1) % K;
        count++;
        signal( mutex );
        signal( notEmpty );
}
```

Producer Process

```
while (TRUE) {

        wait( notEmpty );
        wait( mutex );
        item = buffer[out];
        out = (out+1) % K;
        count--;
        signal( mutex );
        signal( notFull );

        Consume Item;
}
```

Consumer Process

Reader-Writer (e.g. files)

```
while (TRUE) {

    wait( roomEmpty );

     Modifies data

     signal( roomEmpty );
}
```

Writer Process

- Initial Values:
    - roomEmpty = S(1)
    - mutex = S(1)
    - nReader = 0

```
while (TRUE) {

     wait( mutex );
     nReader++;
     if (nReader == 1)
          wait( roomEmpty );
     signal( mutex );

     Reads data

     wait( mutex );
     nReader--;
     if (nReader == 0)
          signal( roomEmpty );
     signal( mutex );

}
```

Reader Process

- mutex used to protect CS of nReader
- writer can be starved

Dining Philosophers

- deadlock: all have left only
- livelock: all unable to pick right
- States: Think, Hungry, Eat

Solution 1: Tanenbaum

```
void takeChpStcks( i )
{
        wait( mutex );
        state[i] = HUNGRY;
        safeToEat( i );
        signal( mutex );
        wait( s[i] );
}
```

```
void safeToEat( i )
{
    if( (state[i] == HUNGRY) &&
        (state[LEFT] != EATING) &&
        (state[RIGHT] != EATING) ) {

        state[ i ] = EATING;
        signal( s[i] );
    }
}
```

```
void putChpStcks( i )
{
        wait( mutex );

        state[i] = THINKING;
        safeToEat( LEFT );
        safeToEat( RIGHT );

        signal( mutex );
}
```

- signals to left and right that chopsticks are ready

Solution 2:

```
void philosopher( int i ){
    while (TRUE){
        Think( );
        wait( seats );
        wait( chpStk[LEFT] );
        wait( chpStk[RIGHT] );
        Eat( );
        signal( chpStk[LEFT] );
        signal( chpStk[RIGHT] );
        signal( seats );
    }
}
```

- Limited Eater: n -1 people for n seats