Midterm, Operating Systems

Jay R Bolton

November 6, 2011

1 **(a)** The overhead of switching processes would be reduced. More specifically, when two processes of the same group are switched, the shared resources (such as the processors' address space) would not need to be saved.

There would need to be a new group state data structure that stores a group ID, a list of processes, the shared address space of those processes, and perhaps some control information and accounting. Switching between groups would require that the group's shared resources are saved.

(b) I don't know what would be different about implementing threads with process groups. Threads are a subset of processes and processes are a subset of their groups. The execution of threads within a process should not affect the resource sharing of processes within their group. Threads would store the stacks, their own PC, and would have access to the process group's shared resources.

Threads could be seen as an application-level parallel to kernel-level process groups. The parent process of a thread stores the shared resource while the threads themselves have their own counters and stacks.

(c) Yes. Let's assume that all processes must be members of a group, even if they are the only ones.

We have to consider two new process switching cases: switching within a group and switching across groups. Groups themselves would need to have "ready," "running," and "suspend" states. For example, switching between processes within the same group will not change the group state and will change the process state as normal. Switching across groups would cause the first group to go from "running" to "ready." On IO blocks, the process would most often switch within the group to keep things fast. We could also suspend entire groups.

2 **(a)** See my attached drawing. P1 and P2 can never overlap, but we can overlap P1/P3 or P2/P3. I overlapped P2 and P3 first then did P1. Total time is 50 minutes.

(b) Throughput = average number of jobs completed per time period T
Let our time period T be 50 minutes:
Throughput = 3
Let our time period T be 5 minutes:
Throughput = .3
Let T be 1 minute:
Throughput = $\frac{3}{50}$

(c) Turnaround time = actual time to complete a job
P2+P3 = 20 minutes
P1 = 30 minutes
P1+P2+P3 = 50 minutes
Average = 16.7

(d) Processor utilization = percentage of time that the processor is active (not waiting)
50% for 15 minutes and 25% for 35 minutes.
= 32.5% for 50 minutes.

(e) See my graph for this. Since multiprogramming only allows concurrency and there are no IO interrupts, there is no true overlap of processes so it's going to take the sum of the durations. Total is 65 minutes.

(f) T = 65 minutes, Throughput = 3
T = 5 minutes, Throughput = $\frac{3}{13} \approx .2$
T = 1 minutes, Throughput = $\frac{3}{65} \approx 4.6$

(g) P1 = 30 minutes, P2 = 20 minutes, P3 = 15 minutes
P1+P2+P3 = 65 minutes
Average = $21\frac{2}{5}$

(h) 100%

3 See my attached table.

4 **(a)**
```
// customer_waiting mutually excludes customers
// barber_waiting mutually excludes the barber

void barber() {
  while(true) {
    sem_wait(customer_waiting);
    cut_hair();
    sem_signal(barber_sleeping);
  }
}

void customer() {
  sem_signal(customer_waiting);
  sem_wait(barber_sleeping);
  get_haircut();
  leave();
}

int main() {
  semaphore customer_waiting = 0;
  binary_semaphore barber_sleeping=1;
  parbegin
    barber();
    while(true) customer();
  parend;
  return 0;
}
```

(b)
```
// seats_mutex prevents deadlock when we have limited seats
  // it mutually excludes free_seats

void barber() {
  while(true) {
    sem_wait(customer_waiting);
    sem_wait(seats_mutex);
    free_seats++;
    cut_hair();
    sem_signal(barber_sleeping);
    sem_signal(seats_mutex);
  }
}

void customer() {
  sem_wait(seats_mutex);
  if(free_seats > 0) {
    free_seats--;
    sem_signal(customer_waiting);
    sem_signal(seats_mutex);
    sem_wait(barber_sleeping);
    get_haircut();
  }
  else {
    sem_signal(seats_mutex);
    leave();
  }
}

int main() {
  semaphore customer_waiting = 0;
  binary_semaphore barber_sleeping=1, seats_mutex=1;
  int free_seats = 5;
```

```
    parbegin
      barber();
      while(true) customer();
    parend;
    return 0;
  }
```

5 See the attached table.